A Practitioner's Guide to Robust Numerical Calculations of Derivatives

an eBook by Larry Trammell

A free and open document

Released: January 2, 2025

ISBN: 979-889901017-0

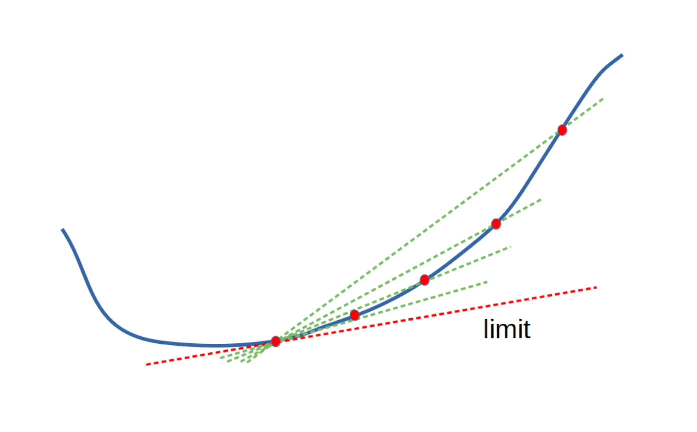
Table of Contents

Introduction: the problem, the challenges

Since this book is about *derivatives*, it is assumed that you have some level of familiarity with *differential calculus* [1], basic *trigonometric functions* [2], and *systems of linear equations and their matrix representations* [3]. If that material seems borderline advanced, or maybe a little faded by passage of time, that is OK. The goal is obtaining practical results, not exposition of mathematical theory. Brief informal reviews are provided each time a concept is brought into the discussion.

We will start at the very beginning.

A *derivative* can be obtained as the limiting slope for a sequence of secant lines constructed between a specified point and a second point, as the second point gets closer and closer to the specified point. Strictly speaking, this is a one-sided derivative. A similar process applied on the other side of the specified point must also yield the same limiting value.



For purposes of numerically *estimating* a derivative value, employing the same strategy is not always possible. We do not always have precise values of the function at arbitrary locations on demand. There could be any number of reasons: maybe because such values are too difficult to compute, or maybe because tabulated points were collected by field measurements.

If we have good reason to believe that the available function values adequately characterize an underlying continuous and differentiable function, we might attempt to construct a continuous and differentiable function that fits the known values well, at least locally. It is then plausible that the derivatives of the approximate model will provide good estimates for the derivatives of the underlying actual function. There is some solid mathematics of *uniform convergence* [4] to back up this idea.

For example, the approximating function might be assumed to be a polynomial. Polynomials are easy to

evaluate, and have continuous derivatives of all orders. Furthermore, it is well known that Newton’s Interpolation Formula [5] can be truncated to form a suitable polynomial. Thus, using a polynomial approximation is an appealing choice, and a good place to start.

We will consider the important special cases for which the following are true.

1. The derivative values are desired at the same points where function values are given. This is typical when values are organized as a file or data array for computer analysis.

2. The abscissa points where function values are known are equally-spaced. This occurs, for example, when data are captured at equal time intervals by sensor electronics.

Just because values are "known" does not mean that they are trustworthy. There are many ways that the “known” function values can be contaminated:

- Values might result from computer calculations which, by necessity, are constrained to a finite number of bits, leading to a *precision error.*

- Valuations may be produced by a convergent process that must be terminated at some point, leaving a *truncation error*.

- “Observed values” might be recorded manually, using only a limited number of significant digits, leaving a *roundoff error*.

- Values might be obtained by measurement, subject to sensor accuracy, quantization noise, and environmental interference.

Derivative estimators should be relatively insensitive to small disturbances in the function's sample values. An estimator with this property is said to be *robust to noise disturbances*, or more simply, *robust*.

The task, then, is to *obtain accurate estimates for the derivative values of the function* given only the *discrete set of imperfect function valuations available.*

This is entirely about appropriate application of mathematical techniques that have been available for a long time; hence, of no academic interest, hence difficult to find in out-of-print technical or scholastic literature, hence not covered by encyclopedic sources such as *Wikipedia*. That is why you need this book.

Formulas based on polynomial approximations

First-order approximation formula

This type of formula for estimating derivative values is obtained by constructing a polynomial model matching the given function values at a finite number of points. The polynomial does not need to be a one-size-fits-all model for the entire data set, and in fact, it is much better to use local approximations custom fit for each location. One way to get such a fit is by Newton's Interpolation Formula [5]. The locations of known function values are presumed to be equally spaced, so that positions in the data can be identified by a local index value *k*, with the point *c* where an approximation is to be evaluated having the central index value of *k = 0*. The relevant neighborhood points used by the estimation formulas are then indexed by *k* in the range -*M* to *M*, where M is appropriately selected. Each unit of index *k* corresponds to one step of length *h* in the domain of the function.

Expressing this mathematically, the equally spaced values of function arguments *x* in the original data set can be represented using the mapping

x = c + k h x = c + {k h}

The corresponding function values y are then given by

y (c − Mh)   y (c − Mh + h)   y (c − Mh + 2 h) ... lbrace matrix{ y(c-Mh) # y(c-Mh+h) # y(c-Mh+2h) # ... # y(c) # ... # y(c+Mh-h) # y(c+Mh) } rbrace   y (c) ...   y (c + Mh − h)   y (c + Mh) lbrace matrix{ y(c-Mh) # y(c-Mh+h) # y(c-Mh +2h) # ... # y(c) # ... # y(c+Mh-h) # y(c+Mh) } rbrace

Since the values of *c*, *M*, and *h* are constant, we can simplify notation and identify the discrete locations by the index variable *k* alone, with the corresponding function values indicated as

y(-M),   *y(-M+1),*   ... *y(0),* ...   *y(M-1),*   *y(M)*

where the step size is now considered to be 1 unit, and location *k=0* is the place where the derivative value is to be estimated. Note that

y ˙ (x) = y ˙ (x (k)) = 1 h · y ˙ (k) dot y(x) = dot y (x(k)) = 1 over h cdot dot y(k)

That is, when we consider *y* as a function of indexing variable *k*, the derivative values calculated must be scaled by *1/h* to return to the units of the original function data.

In the simplest case, Newton’s Interpolation Formula is truncated to only the first two terms.

y (x (0)) = y (x (−1)) + (y (x (0)) − y (x (−1))) 1 ! · h + ... y (x(0)) = y(x(1)) + (y(x(0)) - y(x(1))) over { 1! } cdot h + ...

It is very clear that this describes a line from the function value at -1 to the function value at 0. When this is differentiated, the constant term goes to zero and the *h* variable drops out of the first order term, leaving

y ˙ (x (0)) ≈ (y (x (0)) − y (x (−1))) y ' (x(0)) approx (y(x(1)) - y(x(0)))

or in the simplified notation,

$\dot{y}(0) \approx (y(0) - y(-1))$ y ' (0) approx (y(1) - y(0))
That is, the first-order polynomial approximation to the derivative is simply the differences between consecutive known function values at unit spacing. Very simple… and as you might conjecture, not particularly good. Noise from the two function values will combine and appear directly in the derivative estimates.

Second-order approximation formula

As mentioned in the introduction, the derivative must be the same whether evaluated from the right or left side of a point of interest. The function values at index locations $k = 1$ and $k = -1$ are both just one step away, the same except for the direction of the step. The first-order Newton's Interpolation Formula can be used to obtain an estimate for the reverse-step case as well.

$\dot{y}(0) \approx (y(0) - y(1)) \cdot (-1) = y(1) - y(0)$ y ' (0) approx (y(-1) - y(0)) cdot (-1) = y(0) - y(-1)
The approximations produced by the two cases would be exactly the same if the known function values were completely accurate and obtained from sampling a linear function. Otherwise (and more generally) we should expect that there would be small differences between the two estimates. Without a particular reason to prefer one over the other, maybe a better approximation is obtained by "splitting the difference" and using the average of the estimates obtained from the two sides. Applying this idea yields

$$\dot{y}(0) \approx \frac{y(1) - y(-1)}{2}$$

As it turns out, we could have obtained exactly this same expression by retaining the first three terms from the Newton Formula rather than just the first two. Thus, this formula is actually a second-order approximation that results from matching the known function values at locations *k = -1, 0,* and *+1* with a quadratic polynomial.

Generally speaking, this second-order model is so much better than the first-order model that there is really no point in bothering with the first-order approximation.

Something that initially seems remarkable is that only two of the three known function values contribute to the second-order derivative approximation – the function value at the central location 0 is gone. Yet, we know that when any quadratic function is differentiated, the constant coefficient in that quadratic form disappears, and that disappearance should reflect in the derivative estimator formula.

Suppose that the second-order formula is applied to the function values at locations 0, ½ , and 1. This requires a step size of ½ rather than 1. We can calculate this formula without knowing a function value at location ½. The half-step interval results in a scaling factor of 2. Another scaling factor is introduced when the quadratic curve is differentiated. The extra factors from step scaling and differentiation cancel out. Thus, we can get an estimate for the derivative at the unusual location *k* = ½ from

$$\dot{y}(1/2) \approx 2 \cdot \frac{y(1) - y(0)}{2} = (y(1) - y(0))$$

This is a reasonably good second-order estimate for the derivative value at location ½. But wait... this is exactly the same formula we previously obtained as a first-order estimate for the derivative value at location 0. So which is it?

Actually, both. A first-order estimator is a poor estimate for the derivative at location 0 precisely because it is a relatively better estimate for the value of the derivative at location ½ – at a *shifted location.* Estimation formulas that use data in a balanced manner, with the same number of values before and after the central location of interest, avoid this shifting hazard.

Another important property of the derivative approximation is its anti-symmetry. Notice from the second order estimate at location zero that the coefficients on the y(-1) and y(1) terms are the same except for the opposing signs, while the coefficient at the center location is necessarily zero. This is no accident. If the function values are decomposed into an even-symmetry part and an odd-symmetry part about location 0, the even symmetry parts will contribute nothing to the value of a derivative there.

Extending to higher orders

The motivation for trying polynomial orders higher than 2: improved accuracy. But *are the results really any better?*

First, let’s describe a process that can more generally

identify the polynomials that fit the available function data exactly at a balanced set of abscissa points. Select the polynomial order that you want to fit. For example, if the order is 6, set $N = 6$. The polynomial coefficients to be determined are $P_6, P_5, P_4, \ldots, P_0$.

Select a group of points at which the function values are to be matched, indexed as $-M, -M+1, -M+2, \ldots , -1, 0, 1, \ldots , M-1, M$. The following matrix X collects all of the "power of x" argument terms that will appear in the polynomial as expanded at unit steps from $-M$ to $+M$.

$$
\begin{bmatrix}
(-M)^N & (-M)^{N-1} & (-M)^{N-2} & \cdots & (-M)^3 & (-M)^2 & -M & 1 \\
(-M+1)^N & (-M+1)^{N-1} & (-M+1)^{N-2} & \cdots & (-M+1)^3 & (-M+1)^2 & -M+1 & 1 \\
 & & & \cdots & & & & \\
-2^N & -2^{N-1} & -2^{N-2} & \cdots & -8 & 4 & -2 & 1 \\
-1^N & -1^{N-1} & -1^{N-2} & \cdots & -1 & 1 & -1 & 1 \\
0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\
1^N & 1^{N-1} & 1^{N-2} & \cdots & 1 & 1 & 1 & 1 \\
2^N & 2^{N-1} & 2^{N-2} & \cdots & 8 & 4 & 2 & 1 \\
 & & & \cdots & & & & \\
(M-1)^N & (M-1)^{N-1} & (M-1)^{N-2} & \cdots & (M-1)^3 & (M-1)^2 & M-1 & 1 \\
(M)^N & (M)^{N-1} & (M)^{N-2} & \cdots & (M)^3 & (M)^2 & M & 1
\end{bmatrix}
$$

For a data sequence with $2M+1$ values and polynomial order N, the matrix X will have $2M+1$ rows and $N+1$ columns. The values of the function that the polynomial is

supposed to match can be collected into vector Y having *2M + 1* terms. We can reserve *N+1* locations in a vector P for the polynomial coefficients to be determined. The polynomial coefficients *P(N), P(N-1), …, P(0),* once these are known, will multiply their respective columns in matrix from left to right. Summing scaled columns along each row yields a vector of values as calculated by the model.

The relationship can be represented compactly by the matrix equation

$X \cdot P = Y$ X P = Y

If *2M+1 = N+1*, this matrix is square, and provided that it is *nonsingular* [3] – not a problem in this particular case – the X matrix can be inverted using your favorite mathematical software, producing the relationship

$X^{-1} \cdot Y = P$ B cdot Y = P

Next, we want to differentiate the estimator polynomial to obtain the approximation for the original function's derivatives. We know that when we do this, the P_0 term vanishes. We can disregard the last term of P, hence disregard the last column of matrix X^{-1}. We also know that when the derivative of polynomial P(x) is evaluated at location *x = 0*, any of the terms with a nonzero power of *x* will go to zero, and that means that we don't care about the terms P_2 and beyond. The only term we need to know is P_1, and the formula for that is given by the next to last row in matrix X^{-1}. Extract the terms from that row to obtain the coefficients for the derivative estimator formula.

We have done these calculations for you, and the following table lists these results, for matching polynomials up to order 8 (which, as a practical matter, is as high as you need

to go).

Table of matched polynomial estimators

```
Order 2

 -0.50000000  -0.00000000  0.50000000

Order 4

 0.08333333  -0.66666667  0.00000000  0.66666667
 -0.08333333

Order 6

 -0.01666667  0.15000000  -0.75000000  0.00000000
 0.75000000
 -0.15000000  0.75000000  0.01666667

Order 8

 0.00357143  -0.03809524  0.20000000  -0.80000000
 0.00000000
 0.80000000  -0.20000000  0.03809524  -0.00357143
```

The coefficients you see for the order-2 formula should be no surprise. They match what we found previously using Newton's Formula. To apply that estimator, the three given coefficients will multiply the corresponding function value terms *y(-1)*, *y(0)*, and *y(1)*. For the next higher-order estimator, the coefficients will apply to *y(-2)*, *y(-1)*, *y(0)*, *y(1)*, and *y(2)* … and so forth for higher orders.

Central difference formulas

R. W. Hamming [6] used the *Vandemonde Determinant* to show that the interpolating polynomial is unique, so it doesn't matter what kind of reorganization is applied to the function data values before fitting is done. The underlying result is precisely the same polynomial. All of the various classical interpolation formulas – Newton's difference formulas, Stirling's formulas, Everett's formulas, and so forth [7] – are based on the same polynomial and are equivalent numerically.

But there is one variation that requires special attention, because if you do an academic literature search, it is the one you are going to find. It is called the *method of central differences*. [8]

Central differences are just combinations of known function values. They can be produced in a *difference table*, as illustrated below.

Tabulated calculations of central differences

Δy				
y_{-4}				
$y_{-3} - y_{-4}$				
$\Delta_1 y_{-3} - \Delta_1 y_{-4}$				

$y_{-2} - y_{-3}$ $\delta_2 y_{-2} - \delta_2 y_{-3}$

$\Delta_3 y_{-2} - \Delta_3 y_{-3}$

$y_{-1} - y_{-2}$ $\delta_2 y_{-1} - \delta_2 y_{-2}$

$\Delta_1 y_{-1} - \Delta_1 y_{-2}$

$y_0 - y_{-1}$ $\delta_2 y_0 - \delta_2 y_{-1}$

$\Delta_1 y_0 - \Delta_1 y_{-1}$

$y_1 - y_0$ $\delta_2 y_1 - \delta_2 y_0$

$\Delta_3 y_1 - \Delta_3 y_0$

$y_2 - y_1$ $\delta_2 y_2 - \delta_2 y_1$

$\Delta_3 y_2 - \Delta_3 y_1$

$y_3 - y_2$ $\delta_2 y_3 - \delta_2 y_2$

$\Delta_3 y_3 - \Delta_3 y_2$

$y_4 - y_3$ $\delta_2 y_4 - \delta_2 y_3$

$\Delta_1 y_4 - \Delta_1 y_3$

$y_5 - y_4$

This is actually much simpler than it first appears. The first column is the argument values, which are fixed. The second column lists the corresponding function values to match. To obtain the terms in all of the other columns, simply take two numbers from the preceding column, the term just below and just above, and subtract them. Continue until the table has enough columns and is sufficiently populated for what you need.

The values that you use for the evaluation are the highlighted ones. These values are the *central differences* δ_2 of order 2, δ_4 of order 4, etc., that are associated with evaluations at table location 0.

Once these difference calculations are done, the polynomial approximation formulas for the derivative can be recast into the following form:

$$y'_0 = (1/2)(y_1 - y_{-1}) - (1/12)(\delta_2(y_1) - \delta_2(y_{-1})) + 1/60(\delta_4(y_1) - \delta_4(y_{-1})) - 1/280(\delta_6(y_1) - \delta_6(y_{-1})) + 1/1260(\delta_8(y_1) - \delta_8(y_{-1})) - \ldots$$

Pick the central difference terms out of the table, and apply the indicated multiplier coefficients. This formula can be truncated to the desired number of terms.

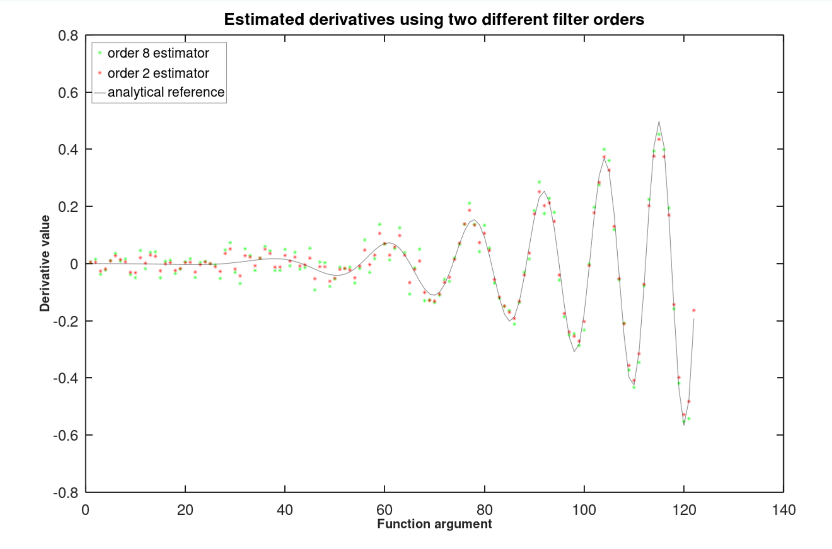
> Congratulations are in order. *If you have covered all of this material so far, you now have earned your degree in numerical derivative computations, as taught at colleges and universities everywhere.*

So, you might ask, why doesn't this book stop here? It is because, under most ordinary circumstances, the results

aren’t very good. While intuitive and effective for the original purpose (tabulating the results of hand calculations), polynomial matching can be considered an underperforming technology.

To show why, let’s see what happens when a polynomial fit estimator is applied to function values affected by low levels of noise. An artificial data set was calculated using a known function corrupted by additive zero-mean white noise. Ordinarily, the function is not known, or you would just differentiate it. But as long as we *do know* what the function and its derivatives are, this provides a means to check the estimator results.

Figure 1 compares results when using a second-order and an eighth-order polynomial fit estimator.



It can be seen that noise in the function data is amplified as it passes through the derivative estimator. Noise effects are particularly troublesome when derivative values are small. Because of this noise, the performance of the order-2 estimator is actually better than the performance of the order-8 estimator! The extra accuracy provided by the higher order is defeated by the noise obscuring its results. Unless you are certain that your function values are very accurate and clean, *there is no point in using polynomial-fit (or central difference) estimators beyond order 2.*

If there is to be any hope in improving the performance of the derivative estimators, it is necessary to address the problem of sensitivity to corruptions in the function data.

Low-order least squares smoothing estimators

Noise can be "reduced" the way that statisticians do it routinely – use more than the minimal number of function values, i.e., more *degrees of freedom* [9], and reap the benefits of “averaging” to reduce “variance.” This suggests the highly plausible strategy of smoothing the data before derivative estimation. Or more efficiently, perform the smoothing *while* estimating the derivatives.

Instead of fitting function values exactly, one effective smoothing strategy is to approximately fit a relatively low-order polynomial. This will not match the data points

exactly, but importantly, it will also not track the noise. A larger number of function values is used to build a matrix formulation as if trying to fit all of the values exactly, producing a “tall and skinny" X matrix, something like the following.

$$\begin{bmatrix} (-M)^4 & (-M)^3 & (-M)^2 & -M & 1 \\ (-M+1)^4 & (-M+1)^3 & (-M+1)^2 & -M+1 & 1 \\ & & \cdots & & \\ 16 & -8 & 4 & -2 & 1 \\ -1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ & & \cdots & & \\ (M-1)^4 & (M-1)^3 & (M-1)^2 & M-1 & 1 \\ (M)^4 & (M)^3 & (M)^2 & M & 1 \end{bmatrix}$$

Five columns are shown in this illustration because a fourth-order polynomial fit is assumed (with the goal of

obtaining a third-order polynomial to approximate the derivatives). The matrix relationship between polynomial coefficients and function values again has the form

$$X P = Y$$

But now, in general, there is no solution P for the polynomial coefficients – *linear algebra* tells us that this hazard can arise for any matrices that have more rows than columns [3]. The best we can get is some kind of approximate fit.

The necessary conditions for the values of the polynomial coefficients P to produce a *least squares* best fit are: [10]

$$(X^T \cdot X)^{-1} \cdot X^T = B$$

$$B \cdot Y = P$$

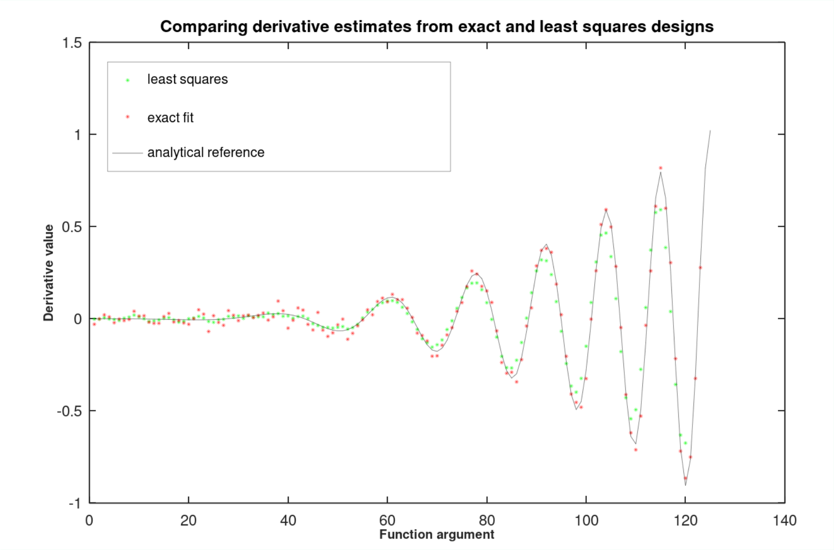
These matrix calculations are no problem if you have appropriate software – which is readily available. The polynomial you get provides a "smoothed" approximation to the known function values, and it is the “smoothing” that suppresses noise. When this P polynomial is differentiated, and the resulting polynomial formula is evaluated at the central argument point 0, an estimator formula is obtained by picking the coefficients from the next to last row of the B matrix, as done for the exact-fit formulas.

Some of these calculations have been done for you, producing the following derivative estimator formulas.

Table of least squares derivative estimators

```
7 terms, order 2 fit

 -0.10714286  -0.07142857  -0.03571429  0.00000000
 0.03571429
 0.07142857  0.10714286

9 values, order 2 fit

 -0.06666667  -0.05000000  -0.03333333  -0.01666667
 0.00000000
 0.01666667  0.03333333  0.05000000  0.06666667

11 values, order 2 fit

 -0.04545454  -0.03636363  -0.02727272  -0.01818181
 -0.00909091
 0.00000000    0.00909091  0.01818181  0.02727272
 0.03636363
 0.04545454

-------

9 values, order 4 fit

 0.72390572  -0.11952862   -0.16245791  -0.10606061
 0.00000000
 0.10606061  0.16245791  0.11952862  0.72390572

11 values, order 4 fit

 0.05827506  -0.05710956  -0.1033411  -0.09770785
 -0.05749806  0.00000000
 0.0574981  0.09770785   0.1033411  0.05710956
 -0.05827506

13 values, order 4 fit

 0.04716117  -0.02747253   -0.06568432   -0.07475857
 -0.06197969  -0.03463203
 0.00000000   0.03463203  0.06197969   0.07475857
  0.06568432  0.02747253
 -0.04716117
```

As good as the ideas might have seemed, these formulas have a different kind of deficiency. We can see this in an example where the 11-term least squares estimator is compared to the 5-term exact polynomial estimator, using the same “known function” test as before.



The least-squares estimator is distinctly better at low frequencies, where derivatives change very slowly. That is the expected effect of the data smoothing. But the accuracy is poor where the derivative values change more rapidly. This hazard is typical of data smoothing approaches: how can you know whether you are removing part of the information you were hoping to capture?

To get some control of the trade-off between noise sensitivity and accuracy, we need some additional tools, which takes us into the next chapter.

Formulas based on filtering

Discrete filtering basics

Let us restrict attention temporarily to a limited class of functions: the trigonometric functions, sine and cosine. If we can analyze how derivative estimation formulas respond to these simple waves, we can use methods of signal analysis theory [11] to decompose a complicated wave shape into multiple sine and cosine functions, perform analysis one wave at a time, and then combine all of the partial results.

We will start by selecting N waveforms, with frequencies such that 1 cycle, 2 cycles, etc. span our data set. The 2 N + 1 waveform functions at “positive and negative frequencies” can be generated by

sin(`π n x / N`)

or by

cos(`π n x / N`)

for *n* in the range -N … +N, and for *x* ranging continuously from `-N to +N.`

Note that:

- When *n = 1*, letting *x* range from -N to +N

continuously will generate one complete waveform cycle.

- When $n = 2$, letting x range from −N to +N continuously will generate two complete waveform cycles. And so forth for $n = 3, 4, 5$, etc. .
- Negative frequencies can be constructed from the positive ones using symmetry.

To work with a finite collection of *x* values sampled from the continuous range, let

```
        x = k N/M
```

and substitute for continuous variable `x`.

```
        sin( [k N/M] π n / N )   =   sin( π k n / M )
```

for *k* in the range `-M` to `M`.

We can note that sine and cosine functions have *phase shift* relationships

$\cos(\pi\cdot x + \pi/2) = -\sin(\pi\cdot x)$ $\sin(\pi\cdot x + \pi/2) = \cos(\pi\cdot x)$

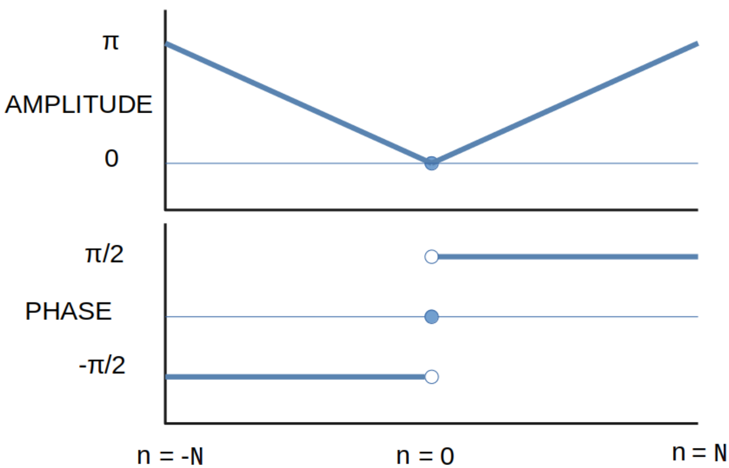
and also the well-known derivative relationships

$\sin'(\pi\cdot x) = \pi\cdot(\cos(\pi\cdot x)) = \pi\cdot(\sin(\pi\cdot x + \pi/2))$ sin'(π·x) = π · (cos(π·x)) = π · (sin(π·x+π/2)) $\cos'(\pi\cdot x) = \pi\cdot(-\sin(\pi\cdot x)) = \pi\cdot(\cos(\pi\cdot x + \pi/2))$ cos'(π·x) = π · (-sin(π·x)) = π · (cos(π·x+π/2))

When evaluated at the discrete waveform locations indexed by *k*

$\sin'(k \cdot \pi \cdot n / M) = \pi \cdot n / M \cdot \sin(k \cdot \pi \cdot n / M + \pi / 2)$ sin'(k · π · n / M) = π · n / M · sin (k · π · n/M + π/2) $\cos'(k \cdot \pi \cdot n / M) = \pi \cdot n / M \cdot \cos(k \cdot \pi \cdot n / M + \pi / 2)$ cos'(k · π· n / M) = π · n / M · cos(k · π · n / M + π/2)

We can summarize the critical information in the form of spectrum graphs displaying the effect that a derivative operation has on a waveform of each frequency: scale the magnitude by an appropriate amount, and shift the phase by an appropriate amount.



Effects of a derivative operator on spectrum

What frequencies are relevant

This is a judgment call of critical importance. Suppose

your data sequence includes as one of its components a trigonometric waveform with a frequency of n = N, as described in the previous section. Samples of a trigonometric waveform at this frequency exhibit a pattern that looks like

W, -W, W, -W, W, -W, …

Is this a kind of pattern you are attempting to quantify using your derivative estimates? Most likely not – you would consider this *high frequency chatter.* This kind of chattering occurs at the limit of what a discrete data sequence can unambiguous represent. (Signal processing experts will say that this is the *Nyquist Limit.* [12])

What about the frequency n = N /2? The pattern that you will observe from this kind of wave is like

W, W, -W, -W, W, W, -W, -W, …

This is the same sort of chattering, except the values occur in alternating pairs. Again, this is unlikely to have any relevance to the derivative values you are trying to estimate.

Similar remarks apply for all of the frequencies between N/2 and N. That is, about half of the spectrum that the derivative response curves can theoretically represent provides no useful information about the derivatives you care about.

Let's propose a criterion for frequency values that should be considered useful.

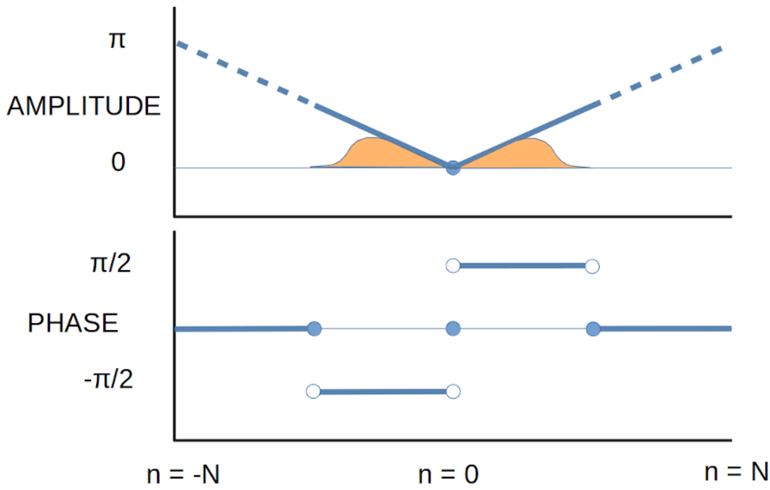
1. As we have just observed, the derivative estimator

should attempt to disregard any frequencies in the upper half of the representable frequency range, from frequencies n = N/2 to n = N.

2. For frequencies from n = 0 to N/5, relevant sine and cosine waves will generate cyclic patterns with perios spanning at least 10 consecutive locations. The derivative estimator should *accurately* characterize these frequencies. (For signal analysis specialists, this translates to “the bandwidth should extend to about 1/5 of the Nyquist Limit.”)

3. The response for other frequencies, the ones in the "transition zone" between n = N/5 and n = N/2, is *unspecified.* As long as there is no sort of wild behavior in this range, almost any kind of filter response can be tolerated.

For comparison, the following graph shows the desired frequency response for a *robust estimator* as compared to the "theoretically perfect” estimator response. Notice how a robust estimator response is consistent with the three rules regarding useful and extraneous frequencies. The white space in the amplitude response, between the theoretical response lines and the horizontal axis, corresponds to the noise that your estimator should try to disregard.



Robust derivative operator

Derivative estimators as filters

The process of applying a polynomial-based estimator to the discrete function data was seen to take the following form:

$$y'(x_0) \approx \sum_{k=-N}^{N} y_k \cdot B_{-k}$$

where the B_k terms are the multiplier coefficients to be applied to the function values, and the derivative estimate is obtained by summing the pairwise products.

As observed previously, derivative estimators were found to exhibit an anti-symmetry property.

$$B_k = -B_{-k}$$

Consequently, an equivalent expression for the estimator evaluation is

$$y'(x_0) \approx -\sum_{k=-N}^{N} y_k \cdot B_{-k}$$

This kind of formulation, with one “forward stepping” index and one “backward stepping” index, corresponds to the definition of a *convolution* operation of the sort that that is ubiquitous in the literature of discrete linear systems and signal processing [13]. A shorthand notation for this operation is

$$Y' = Y * B$$

The set of coefficients B for the convolution operation is called the *convolution kernel*, sometimes called a *filter kernel*.

Highly optimized computer implementations are available for this calculation, which makes it a useful technique for practitioners. But beyond that, this formulation offers some important new insights.

- The derivative estimator is a kind of discrete linear operator that, as it convolves a sequence of function values with a sequence of estimator coefficients, produces discrete derivative estimates at the sample locations.

- Being a discrete filter, a derivative estimator can be described in terms of its frequency response characteristics.

- Derivative estimators do not need a direct relationship to polynomials and do not necessarily need to fit

known data exactly.

Robust max-flat estimators

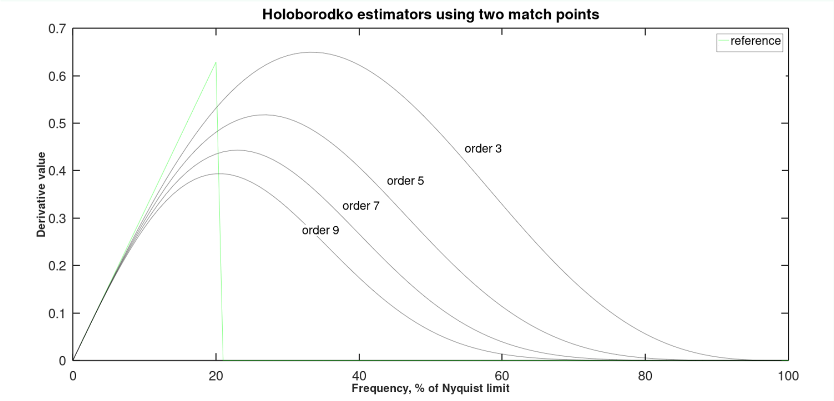
The classic Butterworth filter family [14] has a long history of success, particularly for the design of *low-pass filters* applied to continuous signals. These filters establish a desired gain level at frequency zero, and then attempt to maintain that gain level by forcing the higher derivatives of the gain characteristic to be zero at frequency zero, thereby sustaining "maximal flatness" there. Max-flat digital filters [15] are based on the same ideas, but also have similar and related flatness properties at the high-frequency Nyquist limit.

Pavel Holoborodko proposed a similar criterion for derivative estimation filters, still based on a polynomial form, but configured for the desired spectral properties. His filters have a configurable number of “points of tangency” at locations neighboring zero, typically 2 or 4, to establish the desired frequency response slope of π . An additional constraint artificially forces the magnitude of the response to zero at the Nyquist frequency, suppressing the high “chatter” frequencies. Other terms of the polynomial are selected to force higher derivatives to zero near the central 0 location – making the filter amplitude response track the desired slope as far as it can. Details of the mathematical derivations are not replicated here. You can read the full details of his methods at his Web page. [16]

The following table shows the filter coefficients when the approximation curves match “two points of tangency” at frequencies adjacent to zero.

```
Matching two points of tangency near frequency 0

5 term
 -0.12500000  -0.25000000  0.00000000  0.25000000
 0.12500000

7 term
 -0.03125000  -0.12500000  -0.15625000  0.00000000
 0.15625000
  0.12500000   0.03125000

9 term
 -0.00781250  -0.04687500  -0.10937500
 -0.10937500  0.00000000
  0.10937500   0.10937500
  0.04687500   0.00781250

11 term
 -0.00195313  -0.01562500  -0.05273438
 -0.09375000  -0.10937500
  0.00000000   0.10937500   0.09375000
  0.05273438   0.01562500
  0.00195313
```

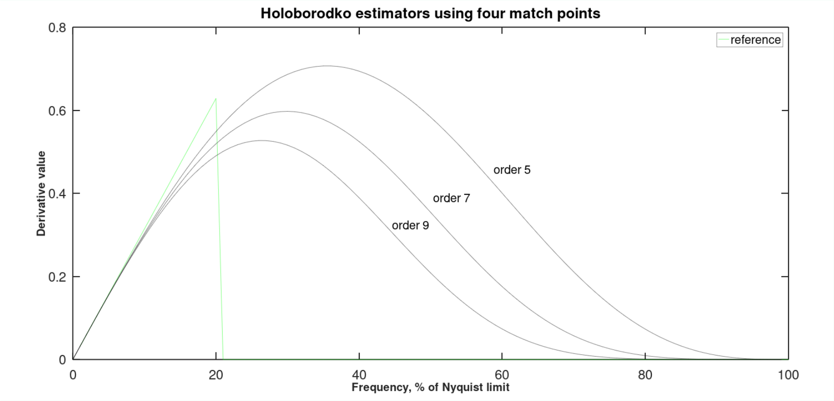
The following plot shows the frequency characteristics of the two-match-point estimators.



The following table shows the resulting filter coefficients when the approximation curves match “four points of tangency” at frequencies adjacent to zero.

```
Matching four points of tangency near frequency 0

7 term
 0.05208333  -0.12500000  -0.40625000  0.00000000
 0.40625000
 0.12500000  -0.05208333

9 term
 0.02083333  0.01041667  -0.16666667  -0.28125000
 0.00000000
 0.28125000  0.16666667  -0.01041667  -0.02083333

11 term
 0.00716146  0.02083333  -0.02539063  -0.16666667
 -0.20963542
 0.00000000  0.20963542   0.16666667   0.02539063
 -0.02083333
 -0.00716146
```

The following plot shows the frequency characteristics of the four-match-point estimators.



The estimators can be seen to track the sloping green line near zero frequency very closely, as intended, while the response decreases to the right of the vertical green line. The two sets of designs are very similar. The two-match formulas having somewhat better noise rejection, while the four-match formulas maintain accuracy through a somewhat wider range. The price of this, however, is that accuracy begins to degrade at about 10% of the Nyquist level, while a somewhat uncomfortable amount of noise leaks through the middle range. The appeal of these formulas is their low frequency accuracy, numerical efficiency, and effectiveness at reducing high-frequency chatter.

FFT based estimator

R. W. Hamming [17] used traditional methods of real analysis, treating the desired spectrum response shape as a continuous function, and using integrals to decompose that function into a series of sine waves. He even took into consideration that “the filter that we design should probably cut off the frequencies at some value w_C.” Regardless of what cutoff might be chosen, he observed that the complete series would be infinite, and truncating an infinite series has consequences. There are two possible options to reduce the undesirable effects. You can either use some form of smoothing for the curve you are approximating, or you can apply some kind of *windowing operation* [18] — a well crafted, term-by-term scaling of the filter coefficients applied with the goal of attenuating extraneous oscillations.

Hamming chose the Lancsos window to reduce the side effects of truncation. No particular reason was given for this choice; his concern was how to design filters, not fine tuning them. The results were good, but not as good as they should be for the amount of computation applied.

We will deviate from Hamming’s scheme in two ways.

1. Instead of doing the calculus in continuous variables, take lots of samples (a large M) along the desired spectrum shape, so that it is well represented, and then obtain the coefficients for the differentiator filter by applying an inverse (reverse) FFT [19] transformation to that sampled spectrum vector. Whether you use the continuous variable analysis or

the discrete FFT analysis actually makes little difference. You end up at essentially the same place, with filter coefficients in the transformed vector, and with the same kinds of disruptions to deal with.

2. Instead of the windowing strategy, try the spectrum shaping approach. When the "theoretical spectrum characteristic" is chopped to zero beyond M/5 and −M/5, these discontinuities cause an FFT to produce artificial "wobbles," known as the *Gibbs phenomenon* [20], a consequence of trying to approximate a discontinuous function with many smooth ones. The wobbles can result in an elongated data sequence, or an elongated spectrum shape. Assigning an artificial but smooth shape to the transition band before constructing the approximation greatly reduces the undesirable wobbles.

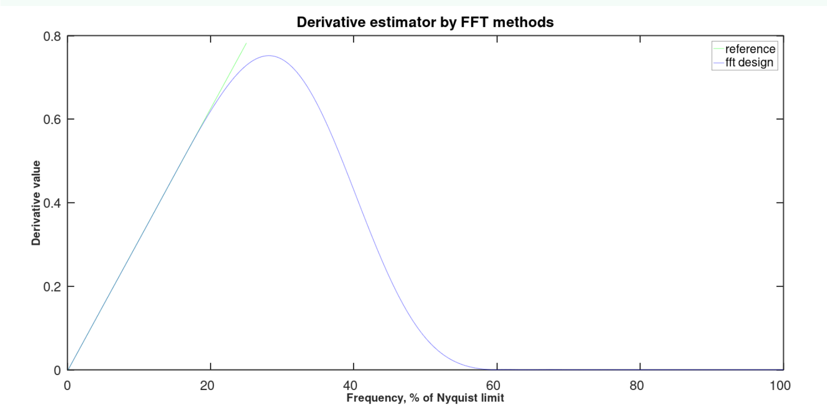
The artificial shape specified for the transition zone between the accurate low band and the suppressed high band can be anything appropriate, because of the many unrestricted response terms available there. Your author has suggested imposing a "transition shape" that begins at the *M/5* frequency band edge and smoothly descends to 0 at the high frequency end of the transition band. This can be interpreted as just another form of *windowing operation*, but with a custom window shape applied. (The selected window shape can be obtained by convolving a "rectangular window" – a finite sequence of constant values – with a "cosine window" – a sequence obtained by sampling a positive half-wave of a cosine function. The resulting window is "flat on top" but has "smoothly descending skirts" at the edges.) After this highly crafted

window is applied to the theoretical spectrum shape, a reverse FFT transform then produces a compact convolution kernel with insignificant remaining chatter. Conventional windowing can then remove that residual chatter with no significant impact on filter accuracy.

Here is a derivative estimator filter kernel obtained using this approach:

```
17-term FFT based derivative estimator

    -0.00476560   -0.00667728    0.00981268    0.03538052   0
    -0.07503981   -0.17324636   -0.15426407    0.00000000   0
     0.17324636    0.07503981   -0.01770558   -0.03538052  -0
     0.00667728    0.00476560
```

Here is a plot of the frequency response achieved, compared to the theoretical curve.



The accuracy is very good, with hardly any tracking error

through to the 20% of Nyquist frequency edge. The transition to zero at high frequencies is smooth but direct, and the high frequency noise rejection is near perfect. Not much mid-range noise leaks through. This filter kernel is four terms shorter, and significantly more accurate, than the one Hamming produced.

What is the catch? This convolution kernel uses 17 terms – a lot. Is all of that really necessary? With minor compromises, perhaps equivalent results could be obtained with a lot less computational effort. But there is nothing to suggest how to do this. We are at the end of this road.

If you have plenty of data in your set of known function points, and the cumbersome filter kernel does not present a computational burden, this estimator can produce excellent results.

Estimators based on least squares spectral approximation

FFT-based designs expend effort equally in every part of the spectrum, in an attempt to deliver *exactly* the response that you specify. Is there a way to say "*as long as the high frequencies are sufficiently attenuated, we really don't care what the filter spectrum looks like there?*" If such a way exists, then numerical processing can be dedicated to achieving good performance using fewer terms.

We have used least squares methods before, with the goal

of fitting a polynomial model to the data. Instead, suppose we apply these methods to the goal of approximating the frequency response of a derivative operator. What sort of approximation would be useful? We know that a Fourier transform represents a sine wave as one spectral point. Similarly, a reverse transform can map a sine waveform in the spectrum back into one data sequence point. This suggests approximating the spectral response using trigonometric functions, which should map back to a compact region of the original data sequence to produce a useful filter kernel.

Select M = 100, so one complete cycle of a waveform is spanned by steps from m = -100 to m = 100. We know that the desired derivative operator's spectrum shape has odd symmetry, so we will employ only sine functions. Where m is negative, the terms of the sine wave differ only in sign, so we simplify by ignoring these terms, reconstructing them later (as needed) using symmetry.

To start the design process, set up a data table.

- The first column is the index m, the numbers ranging from term 0 to 100.
- The second column is for weighting factors, which we will use later, so for now all of its values are set to 1.0.
- The third column is the desired amplitude f for the derivative response. Recall, it will have a slope of π for its first 0.2 * 100 terms – with all of its its other terms set to 0.0 .

- Now add one more column that shows the values of a sine wave for its first half cycle. This will reach its peak value of 1.0 and return to a zero value in the 100 tabulated steps.

m	W	f	Sine 1	...
0	1	0.00000000	0.00000000	
1	1	0.03141076	0.03141076	
2	1	0.06283185	0.06279052	
3	1	0.09424778	0.09410831	
		...		
20	1	0.62831853	0.58778525	
21	1	0.00000000	0.61290705	
		...		
98	1	0.00000000	0.06279052	
99	1	0.00000000	0.03141076	
100	1	0.00000000	0.00000000	

Now add more sine wave columns. The next column will be the sine wave having two times the frequency of the first. The next column will be the sine wave with three times the frequency of the first. And so forth. Continue until you have the number of sine waveforms that you want to use in the approximation. A good place to start is 5 such frequencies (corresponding to an 11-term filter kernel).

m	W	f	Sine 1	Sine2	...	Sine5
0	1	0.00000000	0.00000000	0.00000000		0.0000000
1	1	0.03141076	0.03141076	0.06279052		0.156434
2	1	0.06283185	0.06279052	0.12533323		0.309017
3	1	0.09424778	0.09410831	0.18738131		0.453990
			...			

```
 20   1  0.62831853   0.58778525   0.95105651       0.0000000
 21   1  0.00000000   0.61290705   0.96858316      -0.156434
                          ...
 98   1  0.00000000   0.06279052  -0.12533323       0.309017
 99   1  0.00000000   0.03141076  -0.06279052       0.156434
100   1  0.00000000   0.00000000   0.00000000       0.0000000
```

In the waveform tabulation, each row represents a constraint that we would like to satisfy. That is, we would like some combination of sine wave columns to produce the spectral response levels found in the third column *f*, even though we do not expect this to happen exactly.

Now apply a variation of the least squares technique, called *weighted least squares.* Suppose that we go to tabulation row 2 and change the weighting factor there in column W to a 100. This represents the requirement that every term to the right of the weight factor in that row must be multiplied by 100 before the constraint is used in a least squares fitting problem. The constraint is not changed by this scaling, just as multiplying an equation on the left and right by the same nonzero constant does not change the equality of that equation. However, if that constraint is not satisfied exactly, the resulting fit error will also carry that same scaling factor of 100.   And of course, a least squares fitting problem considers the squares of deviations. In effect, what we have just said is that any fitting error in row 2 is 10000 times more important than the fit errors in other rows.

But on the other hand, if the weighting factor is set to 0, the constraint row would have zero effect on the sum of the squared fitting errors. We might as well not have any such

rows present at all!

Now let's devise some plausible weight factors.

- Sine functions can contribute nothing to the curve fit at location 0. Set the weighting factor in row 0 to 0.0 .
- We want the derivative estimates to be reasonably accurate for the first 20% of the response shape. Leave the weighting factors at 1.0 in rows 1 through 20.
- We learned from the Holoborodko estimators the importance of near-perfect matching near frequency 0 to produce accurate results when derivatives are small. Indicate this importance by resetting the weight factors for rows 1 through 5 to the value 10.
- Rows 21 through 50 represent an unspecified transition zone. We can't apply any penalty when there is no specified target we want to reach. Set all of these weighting factors to 0.0 .
- For rows 51 through 100, we don't care what exactly the responses are, as long as they do not become too large. So give them all a reduced weighting factor of 0.1 .

From this table, construct a matrix representation for applying least squares techniques as follows. Extract sine wave columns 4 through 8, apply the weighting factors row by row, and place these into the matrix X_W . Extract column 2 and also apply the weighting factors to it row by row to produce the weighted response shape vector Y_W .

(Note that rows having a scaling factor of 0 can optionally be removed; they are messy, but neither help nor harm.)

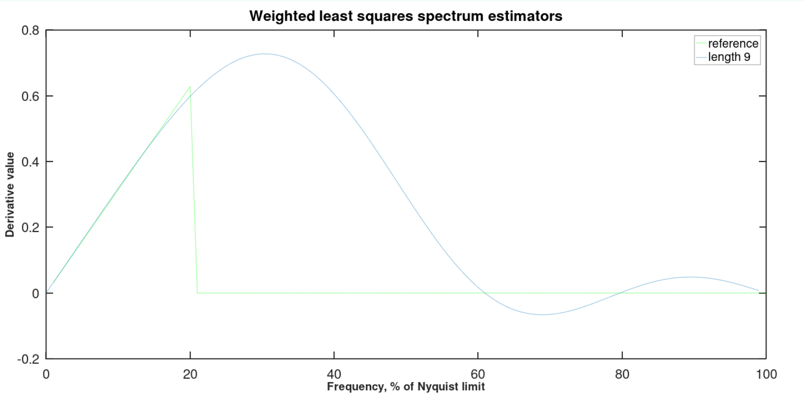
Obtain the multipliers P for the columns, and thus a best match for the target response curve, by solving the necessary conditions for a weighted least squares best solution.

$$A = {X_{w}}^{T} \cdot {X_{w}}$$
$$B = {A}^{-1}$$
$$C = {X_{w}}^{T} \cdot {Y_{w}}$$
$$P = B \cdot C$$

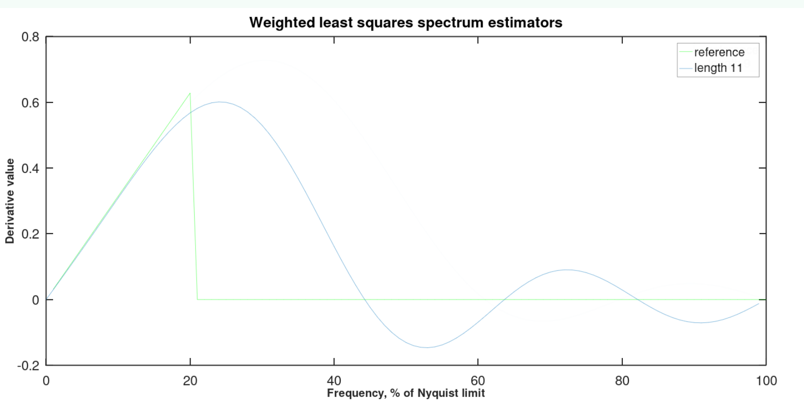
Symmetry properties can then be applied to the P vector multipliers to produce the coefficients of the convolution filter.

Detailed features of the filter response depend on the particular weighting factors selected and exactly where they are applied. There is plenty of art and mystery in this process. But for a few selected scenarios, the tweaking and the heavy computations have been done for you to produce some practical designs. These are presented in the table that follows.

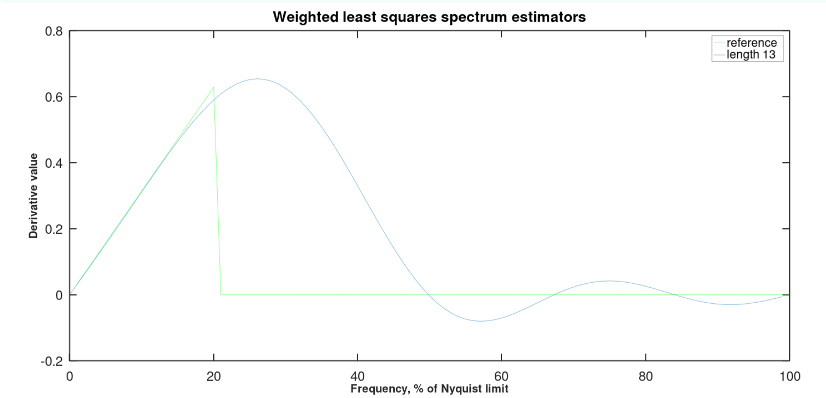
```
Length 9 - Good accuracy and high frequency rejection,
allowing moderate middle-frequency noise

0.04197662 -0.04111849 -0.18244664 -0.18907531  0.00000000
0.18907531  0.18244664  0.04111849 -0.04197662
```




```
Length 11 - Moderate accuracy, vigorous noise rejection
in middle and high frequencies

 0.02874297  0.01236535 -0.07557966  -0.15886784  -0.1356106
 0.00000000  0.13561064  0.15886784   0.07557966  -0.0123653
-0.02874297
```

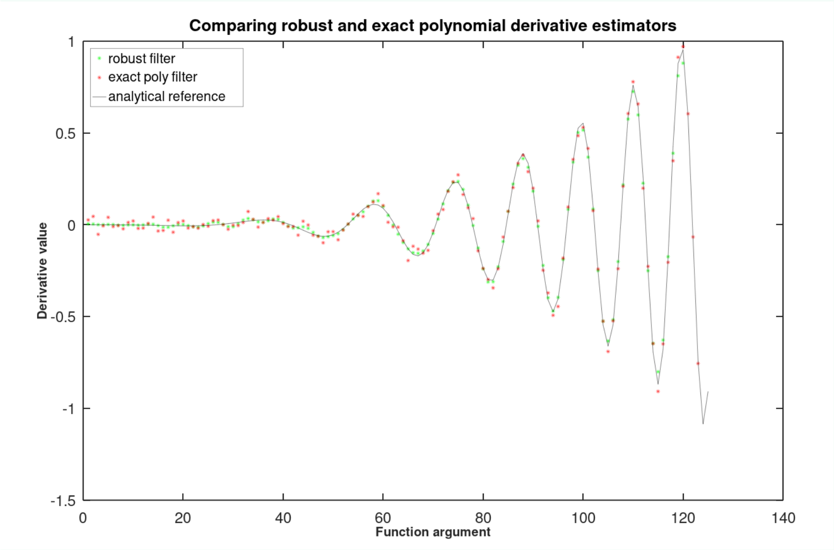


```
Length 13 - Generally good accuracy, good noise rejection

-0.00183293  0.03697227  0.00805005  -0.08655150  -0.154292
-0.12161388  0.00000000  0.12161388   0.15429208   0.086551
-0.00805005 -0.03697227  0.00183293
```



The wobbling in the high-frequency response means that some noise leaks through. That isn't pretty. But noise at these frequencies is multiplied by a factor of roughly 0.1, which means significantly reduced. Whatever leaks through should not matter.

Let's perform one final test, the same "known function" comparison that was used previously. We will pick the 5-term polynomial-match formula, which is known to be quite accurate but vulnerable to noise, and compare to the 11-term spectral-fit estimator, which is middle-of-the-road for accuracy but good for noise rejection.



At low frequencies, the noise rejection of the robust estimator is seen to be excellent, comparing favorably to the least squares smoothing estimator. In the middle frequencies, the tracking accuracy continues to be very good. We would expect the exact polynomial estimator to be more accurate there, but it fails to achieve its potential because of the noise amplification. You can see that the accuracy of the robust estimator declines a tiny amount at the highest frequencies – but that is expected because these frequencies are near the 1/5 of Nyquist frequency band limit, where the response is supposed to begin the transition down toward zero.

The spectral-fit estimators are good general-purpose formulas that can be used with confidence.

Estimators based on min-max spectral approximation

The least squares spectral designs are plenty good for general purpose application. However, least squares designs tend to be aggressive about rejecting the highest frequency noise, while relatively tolerant of noise at middle freqencies. Imagine a situation in which more consistent noise rejection properties are considered important.

We now consider an alternative to *least squares* as a measure for goodness of fit. The mathematics for this is relatively complicated, so we will provide only a general outline of the strategy.

Suppose we start with the data tabulation used in the spectral least squares method, including the weighting factor column. The analysis will need a higher resolution to achieve its objectives, so instead of M = 100 locations, use M = 1000 locations, with correspondingly reduced step so that the number of wave cycles remains the same in each column.

The goal is once again to find multipliers for the sine wave columns so that, in combination, they produce the desired frequency response shape, with the correct slope at low frequencies and roughly zero response at high frequencies. The *f* column specifies the desired response values. When the sine wave columns are multiplied by their respective terms from solution vector P, the sum of these scaled columns MINUS the corresponding *f* column terms should ideally yield zero in every row, achieving a perfect fit.

However, because this is not possible and the fit is not perfect, we can define an extra *slack variable* for each row to make up the difference, resulting in an exact match everywhere.

If we pick any N linearly independent [3] rows out of the tabulation, and ignore all of the other rows, we are left with N constraining equations for the N column multipliers we are trying to determine, to match the N desired values of *f* that correspond to these selected rows. The terms so selected form an NxN square matrix equation that can be uniquely solved[3] to obtain multiplier values. Because this solution produces an exact match to the target values at the selected points, the slack variables at those row locations are zero. Not necessarily so for the rest of the 1000 locations in the table. For these other locations, the curve swings either too high beyond the desired response level, or too low below it. We want these swings to be bounded. That means every slack variable must be smaller than some bound +B, and larger than -B. One very important consequence is that, unlike max-flat methods, frequencies near zero are *not favored for accuracy.*

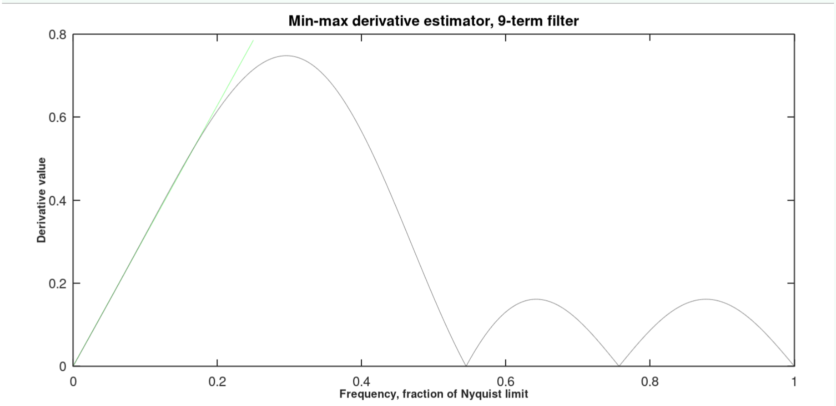
If there is a different selection of N rows, we get different slack variable values for the other rows. Some cases might exhibit deviations that are smaller than those seen previously, producing a better overall fit. In such cases, we can adjust bound B to be smaller. Now, we just need to figure out how to pick the right set of rows so that the corresponding P vector produces the smallest possible bound B. This particular P vector minimizes a bound on the maximum deviations… hence “min-max.” The problem of identifying the proper row locations is addressed by the

Dantzig algorithm [21], which optimizes a linear cost-objective function under linear inequality constraints.

Deviations from the desired reference response are calculated for each row *after* weighting factors are applied. Weighting factors are fixed at 1.0 for the noise band, set to a larger value for the “accurate band,” and set to zero in the transition zone. What really matters is the ratio between the weighting factors for high and low bands. For example, if the accurate band’s weighting factors are 40, a wobble of 0.01 at the low frequency end of the spectrum would be considered to balance a wobble of 0.40 at the high frequency end of the spectrum. As with the least squares designs, certain levels of craftsmanship and fortune are required to select the “best” band edge locations and weighting factors.

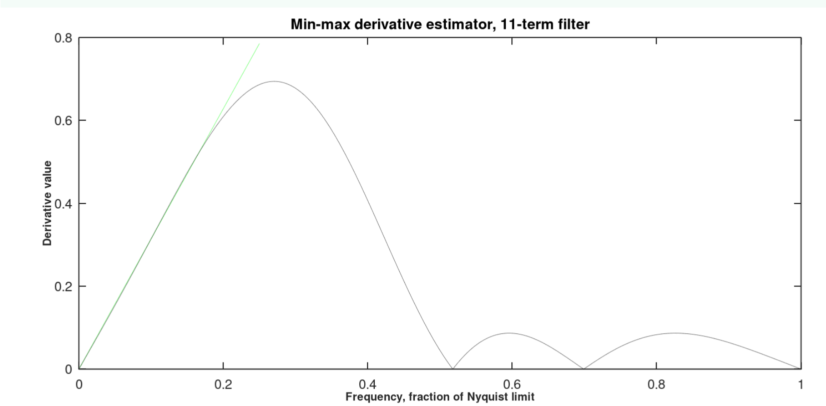
The following listing provides some estimator designs that have been calculated for you.

```
Length 9 - Peak noise gain 0.15571

 0.06684721  -0.08215640  -0.17948715  -0.16197644  0.000000
 0.16197644   0.17948715   0.08215640  -0.06684721
```



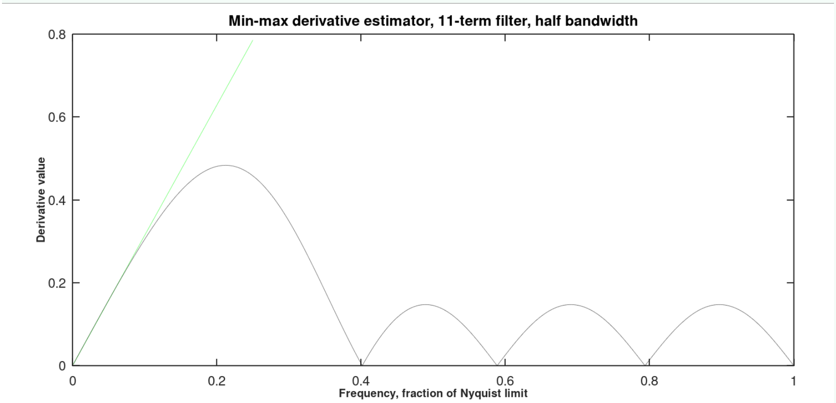
Length 11 - Peak noise gain 0.08680

```
 0.02551581   0.03193725  -0.09409916   -0.15803250  -0.142
 0.00000000   0.14224824   0.15803250    0.09409916  -0.031
-0.02551581
```



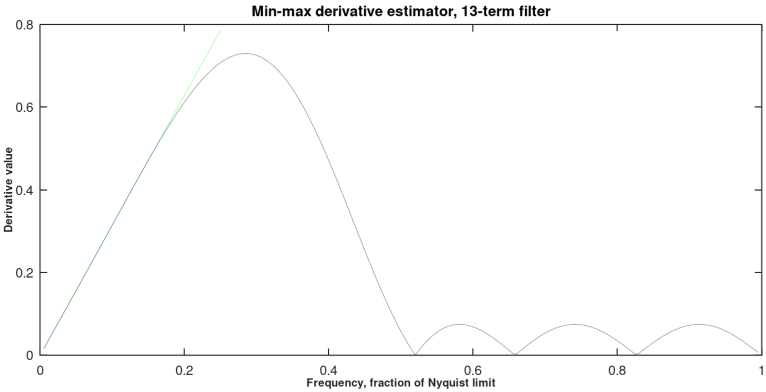
Length 11 - Half band, extended rejection - Peak noise gain

```
 0.04824546  -0.06208660  -0.08510025 -0.09173013  -0.06074
 0.00000000   0.06074938   0.09173013   0.08510025  0.06208
-0.04824546
```



Length 13 - Peak noise gain 0.07896

```
-0.01472205   0.04912907  0.01836830  -0.07708223  -0.17190
-0.15017757   0.00000000  0.15017757   0.17190549  .0770822
-0.01836830  -0.04912907  0.01472205
```


Min-max derivative estimator, 13-term filter
Derivative value
Frequency, fraction of Nyquist limit
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

Are we there yet?

In one sense, no. There are other approaches that could be considered, and there are more things to learn about the practical application of all of these methods. Methods that actually work still depend on too much craftsmanship, while mathematically elegant methods are not competitive.

But in another sense, yes. You now have a number of effective tools, with information about how and when to use them. To summarize:

- Polynomial fitting and central differences estimators: the theoretical ideal, but applicable only for clean accurate data and discussions with the professors.
- Least squares smoothing estimators: a nice idea with good noise reduction, but too inaccurate generally.
- Max-flat estimators: efficient processing, accuracy excellent at low frequencies but with some compromises mid-range.
- FFT estimator: excellent accuracy, excellent noise rejection, but cumbersome.
- Least squares spectral estimators: generally good accuracy and noise rejection, moderate computational efficiency, suggested for general application.
- Min-max spectral estimators: slightly less accurate than least squares spectral estimators, but with better

control over noise attenuation.

Bibliography

These are sources that I referenced for preparing this book. Most of these are quite generic, so if you can't locate them, don't worry, there are many alternative sources that you will probably find suitable.

[1] Lamb, Horace, “An Elementary Course of Infinitesimal Calculus” second edition, Cambridge University Press, 1902; see chapters 1 and 2. Available at https://www.archive.com/.

[2] Nicholas, Jackie, and Adamson, Peggy, “Introduction to Trigonometric Functions”, Mathematics Learning Center at University of Sydney, 1998; available online at https://www.sydney.edu.au/content/dam/students/documents/mathematics-learning-centre/introduction-trigonometric-functions.pdf .

[3] Schneider, Hans, and Barker, Phillip, “Matrices and Linear Algebra,” Hold, Reinhart and Winston, 1968

[4] See https://en.wikipedia.org/wiki/Uniform_convergence

[5] See https://en.wikipedia.org/wiki/Newton_polynomial

[6] Hamming, R. W., “Numerical Methods for Scientists and Engineers” second edition, McGraw Hill Books, 1973. See section 14.3.

[7] Abramowitz, Milton, and Stegun, Irene, editors, “Handbook of Mathematical Functions,” Dover Publications, 1970, see chapter 25.

[8] Froberg, Carl-Erik , “Introduction to Numerical Analysis,” 1964 , Addison Wesley Publishing. See Chapter 9.

[9] Sen, Ashish, and Srivastava, Muni, "Regression Analysis: Theory, Methods, and Applications”. See section 2.7 regarding variance in multiple linear regression models.

[10] Kennedy, William J., and Gentle, James E. , “Statistical Computing,” 1980, Marcel Dekker Inc. See chapter 8 regarding multiple linear regression calculations.

[11] Papoulis, Athanasios, “Signal Analysis,” 1977, McGraw Hill Books, see Chapter 3.

[12] Papoulis, Athanasios, “Signal Analysis,” 1977, McGraw Hill Books, see section 5-1.

[13] Papoulis, Athanasios, “Signal Analysis,” 1977, McGraw Hill Books, see section 1-1.

[14] Pashtoon, Nazir A., in “Handbook of Digital Signal Processing,” 1987, Academic Press Inc. See section 4.VI.A.

[15] Vaidyanathan, P. P., in “Handbook of Digital Signal Processing,” 1987, Academic Press Inc. See section 2.V.

[16] Holoborodko, Pavel, see the Web site at http://www.holoborodko.com/pavel/numerical-methods/numerical-derivative/smooth-low-noise-differentiators/

[17] Hamming, R. W. , "Digital Filters," 1977, Prentice-Hall Inc. See section 6.6

[18] Vaidyanathan, P. P., in "Handbook of Digital Signal Processing," 1987, Academic Press Inc.   See section 2.III

[19] Elliot, Douglas F., in "Handbook of Digital Signal Processing," 1987, Academic Press Inc. See chapter 7.

[20] Hamming, R. W. , "Digital Filters," 1977, Prentice-Hall Inc. See section 5.2.

[21] Murty, Katta, "Linear and Combinational Programming," 1976, John Wiley and Sons.   See chapters 1, 2, and 10.

Addendum

There is no *a priori* reason for you to believe everything you read in this book. The proof is in the doing. So that you can demonstrate it all for yourself, executable scripts are provided in this supplement. These were used to generate all of graphs and calculated coefficients presented in this book. Simply identify the script sections relevant to your projects, and cut-and-paste these into text files on your computer system under the "m-file" names indicated. Then you can then run them using the free and open source Octave software: switch to the directory or folder where the files are placed, run Octave, and then tell Octave to run the command which is the file name less the ".m" decoration.

I hope you will go further than that, and modify the scripts so that the methods are tested using your actual data. Or, take it to the next level, and tune the filter designs to even better match your application requirements.

--- polyderivs.m ---

```
# This script solves for the coefficients of derivative estim
# based on an exact fit polynomial match for a range of estim
#

format long
```

```
'order 2'
mat = [ 1  -1  1 ;  ...
        0  0  1 ;   ...
        1  1  1]
mmat=mat^-1;
poly = mmat(2,:)

'order 4'
mat = [ 16  -8  4  -2  1 ; ...
        1   -1  1  -1  1 ; ...
        0   0   0   0  1 ; ...
        1   1   1   1  1 ; ...
        16   8  4   2  1 ]
mmat=mat^-1;
poly = mmat(4,:)

'order 6'
mat = [ 729   -243   81   -27   9  -3   1 ; ...
        64    -32    16    -8   4  -2   1 ; ...
        1      -1     1    -1   1  -1   1 ; ...
        0      0      0     0   0   0   1 ; ...
        1      1      1    1    1   1   1 ; ...
        64     32    16     8   4   2   1 ; ...
        729   243    81    27   9   3   1 ]
mmat=mat^-1;
poly = mmat(6,:)

'order 8'
mat = [ 65536  -16384  4096  -1024    256   -64   16    -4
        6561   -2187    729   -243     81   -27    9    -3
        256     -128     64    -32     16   -8     4    -2
        1       -1       1      -1      1   -1     1    -1
        0       0        0       0      0    0     0     0
        1        1       1       1      1    1     1     1
        256      128     64     32     16    8     4     2
        6561    2187    729    243     81    27    9     3
        65536  16384   4096   1024    256    64   16     4
mmat=mat^-1;
poly = mmat(8,:)
```

--- testexact.m ---

```
# Compare the response of two exact polynomial fit estimators
# central differences) when applied to the same noisy signal.
```

```
#
Npts = 125
puredat = zeros(Npts,1);
truedrv = zeros(Npts,1);

# Construct the pure analytical function sequence and the sam
# zero mean white noise.
for iv=1:Npts
  arg= pi * iv * iv / 1200 + pi;
  puredat(iv) =  sin(arg) * iv*iv/16000;
  noisdat(iv) = puredat(iv) + (rand(1)-rand(1))*0.08 ;
  truedrv(iv) = pi * 2 * iv / 1200 * cos(arg) * iv*iv/16000 +
               sin(arg) * iv * 2 / 16000 ;
end

#plot(1:Npts, puredat, 'b', 1:Npts, noisdat, 'r', 1:Npts, zer
#title("True and corrupted function data sequences")

# Note that octave uses filter kernels in the REVERSE ORDER..
# convolution operation.  Also note that compensation is need
# that the 'filter' command produces.
kernel2 = [ 1/2   0    -1/2 ];
kernel4 = [ 1/12  -2/3    0   2/3  -1/12 ];
kernel6 = [ 1/60  -3/20  3/4  0  -3/4  3/20  -1/60 ];
kernel8 = [ -1/230  4/105  -2/10  8/10  0   -8/10  2/10  -4/1

# Apply the estimation filters to the noisy data sequence
estdrv2 = filter(kernel2,[1], noisdat);
estdrv4 = filter(kernel4,[1], noisdat);
estdrv6 = filter(kernel4,[1], noisdat);
estdrv8 = filter(kernel8, [1], noisdat);
plot(1:Npts, truedrv, 'k')

# Compare the results
plot(1:Npts-4, estdrv8(5:Npts), 'g*',  1:Npts-3, estdrv2(2:Np
set(gca, "linewidth", 2, "fontsize", 24)
xlabel("Function argument","fontsize",20,"fontweight","bold")
ylabel("Derivative value","fontsize",20,"fontweight","bold")
title("Estimated derivatives using two different estimate ord
legend("order 8 estimator", "order 2 estimator","analytical r
```

--- lsderivs.m ---

```
# This script solves for the coefficients of least-squares sm
# for order 2 and order 4 polynomial fits and various filter
#
format long

'order 2, 7 terms'
mat = [ 9  -3  1 ; ...
        4  -2  1 ;  ...
        1  -1  1 ;   ...
        0  0  1 ;   ...
        1  1  1 ;   ...
        4  2  1 ;   ...
        9  3  1  ]
model = (mat' * mat)^-1 * mat'
poly = model(2,:)


'order 2, 9 terms'
mat = [ 16  -4  1 ; ...
        9  -3  1 ; ...
        4  -2  1 ;  ...
        1  -1  1 ;   ...
        0  0  1 ;   ...
        1  1  1 ;   ...
        4  2  1 ;   ...
        9  3  1 ;   ...
        16 4  1  ]
model = (mat' * mat)^-1 * mat'
poly = model(2,:)


'order 2, 11 terms'
mat = [ 25  -5  1; ...
        16  -4  1; ...
         9  -3  1 ; ...
         4  -2  1 ;  ...
         1  -1  1 ;   ...
         0  0  1 ;   ...
         1  1  1 ;   ...
         4  2  1 ;   ...
         9  3  1 ;   ...
         16 4  1 ;
         25 5  1  ]
model = (mat' * mat)^-1 * mat'
poly = model(2,:)
```

```
'order 2, 13 terms'
mat = [ 36  -6  1 ;
        25  -5  1 ; ...
        16  -4  1 ; ...
         9  -3  1 ; ...
         4  -2  1 ;  ...
         1  -1  1 ;   ...
         0  0  1 ;   ...
         1  1  1 ;   ...
         4  2  1 ;   ...
         9  3  1 ;   ...
         16 4  1 ;
         25 5  1 ;
         36  6 1  ]
model = (mat' * mat)^-1 * mat'
poly = model(2,:)


# -----------------


'order 4, 9 terms'
mat = [  256  -64   16  -4  1 ; ...
         81   -27   9   -3  1 ; ...
         16    -8   4   -2  1 ;  ...
          1    -1   1   -1  1 ;   ...
          0     0   0    0  1 ;   ...
          1     1   1    1  1 ;   ...
          16    8   4    2  1 ;   ...
          81   27   9    3  1 ;  ...
         256   64  16    4  1  ]
model = (mat' * mat)^-1 * mat'
poly = model(4,:)


'order 4, 11 terms'
mat = [  625 -125   25  -5  1 ; ...
         256  -64   16  -4  1 ; ...
         81   -27   9   -3  1 ; ...
         16    -8   4   -2  1 ;  ...
          1    -1   1   -1  1 ;   ...
          0     0   0    0  1 ;   ...
          1     1   1    1  1 ;   ...
          16    8   4    2  1 ;   ...
          81   27   9    3  1 ;  ...
```

```
         256   64  16    4  1 ; ...
         625   125  25    5  1    ]
model = (mat' * mat)^-1 * mat'
poly = model(4,:)


'order 4, 13 terms'
mat = [ 1296 -216   36  -6  1 ; ...
         625 -125   25  -5  1 ; ...
         256  -64   16  -4  1 ; ...
         81   -27   9   -3  1 ; ...
         16    -8   4   -2  1 ;  ...
          1    -1   1   -1  1 ;   ...
          0     0   0    0  1 ;   ...
          1     1   1    1  1 ;   ...
          16    8   4    2  1 ;   ...
          81   27   9    3  1 ;  ...
         256   64  16    4  1 ; ...
         625   125  25   5  1 ; ...
        1296   216  36   6  1 ]

model = (mat' * mat)^-1 * mat'
poly = model(4,:)
```

--- testlls.m ---

```
# Script to compare an exact fit derivative estimator to a le
# derivative estimator when applied to the same known functio
#
# Note that the terms in Octave's filter kernel are REVERSED.
# manner of a convolution operator.
#
Npts = 125
puredat = zeros(Npts,1);
truedrv = zeros(Npts,1);

# Construct the original and noisy curve versions
for iv=1:Npts
  arg= pi * iv * iv / 1200 + pi;
  puredat(iv) =  sin(arg) * iv*iv/10000;
  noisdat(iv) = puredat(iv) + (rand(1)-rand(1))*0.08 ;
  truedrv(iv) = pi * 2 * iv / 1200 * cos(arg) * iv*iv/10000 +
               sin(arg) * iv * 2 / 10000 ;
```

```
end

plot(1:Npts, puredat, 'b', 1:Npts, noisdat, 'r', 1:Npts, zero
title("True and corrupted function data sequences")

# Apply an exact-fit derivative estimator, and independently,
# derivative estimator to the same noisy sequence.
kernelx = [ -0.083333333   0.666666666   0.00000000  -0.66666
kernells = [-0.05827506    0.0571096    0.1033411   0.0977078
  -0.0574981   - 0.09770785    -0.1033411    -0.0571096   0.0

# Apply the estimation filters to the noisy data sequence
estdrvx = filter(kernelx,[1], noisdat);
estdrvlls = filter(kernells, [1], noisdat);

# Compare the results on a plot
plot(1:Npts-5, estdrvlls(6:Npts), 'g*',  1:Npts-2, estdrvx(3:
set(gca, "linewidth", 2, "fontsize", 24)
xlabel("Function argument","fontsize",20,"fontweight","bold")
ylabel("Derivative value","fontsize",20,"fontweight","bold")
title("Comparing derivative estimates from exact and least sq
legend("least squares", "exact fit","analytical reference", "
```

--- holoboodko.m ---

```
# Plot the frequency reponse curves of the Holoborodko first 
#
clear
Nspect = 200
Nplot  = Nspect/2 ;
Nideal = Nplot*0.2;

"Holoborodko estimators using two match points"
# Definitions of Holoborodko max-flat estimators
coeff2ord3 = [ -0.12500000   -0.25000000   0.0  0.25000000  0
coeff2ord5 = [ -0.03125000   -0.12500000   -0.15625000  0.0
    0.03125000 ] ;
coeff2ord7 = [ -0.00781250  -0.04687500   -0.10937500   -0.10
   0.10937500   0.04687500    0.00781250  ] ;
coeff2ord9 = [ -0.00195313   -0.01562500   -0.05273438   -0.0
  0.0   0.08203125   0.09375000   0.05273438   0.01562500   0

# Reserve storage for spectrum calculations
```

```
workr = zeros(1,Nspect);
worki = zeros(1,Nspect);
workc = zeros(1,Nspect);
ideal = zeros(1,Nplot);
freq  = zeros(1,Nplot);
for term=1:Nplot
  freq(term) = term-1;
end
for term=1:Nideal
  ideal(term) = freq(term+1)*pi/Nplot;
end

# Obtain filter responses by FFT analysis
worki(1:3) = coeff2ord3(3:5);
worki(Nspect-1:Nspect) = coeff2ord3(1:2);
workc = workr+worki*j;
spect2ord3 = fft(workc);

worki(1:4) = coeff2ord5(4:7);
worki(Nspect-2:Nspect) = coeff2ord5(1:3);
workc = workr+worki*j;
spect2ord5 = fft(workc);

worki(1:5) = coeff2ord7(5:9);
worki(Nspect-3:Nspect) = coeff2ord7(1:4);
workc = workr+worki*j;
spect2ord7 = fft(workc);

worki(1:6) = coeff2ord9(6:11);
worki(Nspect-4:Nspect) = coeff2ord9(1:5);
workc = workr+worki*j;
spect2ord9 = fft(workc);

# Display filter characteristics for the two-match estimators
plot(1:Nplot, ideal(1:Nplot), 'g',  freq(1:Nplot),spect2ord3(
  freq(1:Nplot),spect2ord5(1:Nplot),'k',  freq(1:Nplot),spect
  freq(1:Nplot),spect2ord9(1:Nplot),'k' )
set(gca, "linewidth", 2, "fontsize", 24)
xlabel("Frequency, % of Nyquist limit","fontsize",20,"fontwei
ylabel("Derivative value","fontsize",20,"fontweight","bold")
title("Holoborodko estimators using two match points")
legend("reference", "order 3", "order 5", "order 7", "order 9
pause(20)

# -----------
```

```
# Repeat analysis for the 4-match variation
workr = zeros(1,Nspect);
worki = zeros(1,Nspect);
workc = zeros(1,Nspect);

"Holoborodko estimators using 4 match points"
coeff4ord5 = [ 0.05208333      -0.12500000    -0.40625000   0
   -0.05208333   ] ;
coeff4ord7 = [ 0.020833333    0.01041667    -0.16666667   -0.
    0.16666667  -0.01041667   -0.020833333] ;
coeff4ord9 = [ 0.00716146   0.02083333   -0.02539063   -0.166
    0.0  0.20963542   0.16666667   0.02539063  -0.02083333

# Obtain filter responses by FFT analysis
worki(1:4) = coeff4ord5(4:7);
worki(Nspect-2:Nspect) = coeff2ord5(1:3);
workc = workr+worki*j;
spect4ord5 = fft(workc);

worki(1:5) = coeff4ord7(5:9);
worki(Nspect-3:Nspect) = coeff2ord7(1:4);
workc = workr+worki*j;
spect4ord7 = fft(workc);

worki(1:6) = coeff4ord9(6:11);
worki(Nspect-4:Nspect) = coeff2ord9(1:5);
workc = workr+worki*j;
spect4ord9 = fft(workc);

# Display filter characteristics for the four-match estimator
plot(1:Nplot, ideal(1:Nplot), 'g',  ...
  freq(1:Nplot),spect4ord5(1:Nplot),'k',  freq(1:Nplot),spect
  freq(1:Nplot),spect4ord9(1:Nplot),'k' )
set(gca, "linewidth", 2, "fontsize", 24)
xlabel("Frequency, % of Nyquist limit","fontsize",20,"fontwei
ylabel("Derivative value","fontsize",20,"fontweight","bold")
title("Holoborodko estimators using four match points")
legend("reference", "order 5", "order 7", "order 9")
```

--- fftestim.m ---

```
% Script to construct a first derivative approximation filter
% using FFT methods.  This uses the 'platcos.m' script to bui
% custom window shape, and the kaiser function from the 'sign
```

```
% package for cleaning up the tails of the filter kernel.
%
clear
format long
pkg load signal

% Design parameters for windowing
match   = 350;
transit = 64;
nfilt   = 21;
alpha   = 6.2;

% FFT properties
blksize = 2000;
nhfilt  = floor(nfilt/2);

% Build the theoretical ideal spectrum for a derivative opera
spect = zeros(1,blksize);
for term=2:blksize/2
  ideal(term)=((term-1)*pi*2/blksize);
  spect(term) = -j * ideal(term);
  spect(blksize+2-term)=-spect(term);
end

% Window the ideal characteristic to desired spectrum band
wind = platcos(match,transit,blksize);
spect = spect .* wind;

% Design filter
seq = real(ifft(spect));
filter(1:nhfilt)=seq(blksize+1-nhfilt:blksize);
filter(nhfilt+1:nfilt)=seq(1:nhfilt+1);
wind = kaiser(nfilt,alpha);
filter = filter' .* wind

% Verify design by calculating its frequency response
[actual,w] = freqz(filter,[1],1000);
actual = abs(actual');

% Plot the resulting design
figure(1)
xargs = (1:1000)/10;
plot(xargs(1:250),ideal(1:250),'g', xargs,actual(1:1000),'b')
set(gca, "linewidth", 2, "fontsize", 24)
xlabel("Frequency, % of Nyquist limit","fontsize",20,"fontwei
ylabel("Derivative value","fontsize",20,"fontweight","bold")
```

```
legend("reference", "fft design")
title('Derivative estimator by FFT methods')

% Optional display, verify correspondence analytically
% figure(2)
% diff=actual-ideal;
% plot(1:100,diff(1:100),'r')
% title('Fit error')
```

--- platcos.m ---

```
function window = platcos(nplat,trans,total)
# This is a subprogram used by the fftestim script.
#
# Construct a symmetric window vector of length 'total'.  The
# has a flat central 'plateau' and smooth slopes down to zero
# This is intended for shaping broad transitions between reta
# rejected bands in a frequency spectrum.
# - The first central term is always 1.
# - 'nplat' terms that follow also have value 1
# - After the platform, 'trans' steps follow a cosine shape t
# - The rest of the terms are zero.
# - Replicate the rest of the window using even symmetry.

window = zeros(1,total);
window(1)=1;
window(2:nplat+1) = ones(1,nplat);
window(total-nplat+1:total) = ones(1,nplat);
for term=1:trans
   window(term+nplat+1) = 0.5*(1 + cos((term)*pi/trans));
   window(total-term-nplat+1) = window(term+nplat+1);
end
```

--- lsspectfit.m ---

```
# Derive a filter approximating an ideal robust filter estima
# using a weighted least squares method.  The resulting frequ
# be closely approximated as a composition of just a few sine
# Fourier analysis).
#
```

```
Nterms = 200
Nhalf = Nterms/2;
Nmid = Nhalf+1;
format long

# Filter design parameters for length 9
Ncoeffs = 6;  Nmatch = 7;  Nfit   = 10;  Nfloor = 54;  matchl

# Filter design parameters for length 11
# Ncoeffs = 7;  Nmatch = 9;  Nfit   = 16;  Nfloor = 32;  matc

# Filter design parameters for length 13
# Ncoeffs = 8;  Nmatch = 12;  Nfit   = 20;  Nfloor = 40;  mat

# Reserve working storage
xmat = zeros(Nhalf,Ncoeffs);
ymat = zeros(Nhalf,1);

# Build weight vector.  Note that row 0 is not present for th
weights = zeros(Nhalf,1);
for term = 1 : Nmatch
   weights(term) = matchlev;
end
for term = Nmatch+1 : Nfit
   weights(term) = 1.0;
end
for term = Nfloor : Nhalf-1
   weights(term) = noiselev;
end
weights(Nhalf) = highlev;

# Populate X matrix with expansions of sine functions
wxmat = xmat;
for col = 1:Ncoeffs
  for row = 1:Nhalf
     xmat(row,col) = sin(col*row*pi/(2*Nhalf));
     wxmat(row,col) = xmat(row,col) * weights(row);
  end
end

# Calculate the reference response shapes
ideal = zeros(Nhalf,1);
rhs = ideal;
for term = 1:Nhalf/5
  ideal(term) = term * pi / Nhalf;
  rhs(term) = ideal(term) * weights(term);
```

```
end

# Construct the least squares best-fit appoximation
A = wxmat' * wxmat;
Ainv = A^(-1);
parms = Ainv * (wxmat' * rhs);
fit = xmat*parms;
# figure(1)
# plot(1:Nhalf, ideal, 'go',  1:Nhalf, fit, 'k.')
# title("Spectrum approximation")

# Represent this spectrum shape as an block
fftblock = zeros(1,Nterms);
fftblock(1) = 0.0;
for term = 2 : Nhalf+1
  fftblock(term) = -fit(term-1) * j;
  fftblock(Nterms+2-term) = -fftblock(term) ;
end

# Compute the corresponding kernel sequence
conv = ifft(fftblock);
resp = real(conv);
figure(2)
#plot(1:Nterms, resp, 'k')
#title("Raw convolution")

# Clean out insignificant terms
wresp = resp;
for term = Ncoeffs : Nterms-Ncoeffs+2
  wresp(term) = 0.0;
end
wresp(1:Ncoeffs)

# Verify the response characteristic of the resulting filter
spect = fft(wresp);
resp = -imag(spect);
plot(1:Nhalf, ideal(1:Nhalf), 'g', 0:Nhalf-1, resp(1:Nhalf))
set(gca, "linewidth", 2, "fontsize", 24)
xlabel("Frequency, % of Nyquist limit","fontsize",20,"fontwei
ylabel("Derivative value","fontsize",20,"fontweight","bold")
title("Weighted least squares spectrum estimator characterist
legend("reference", "length 9")
```

--- testrobust.m ---

```
# Compare the performance of a generic robust estimator obtai
# fitting methods to a classic central differences estimator.
#
# The data are generated by a known function with white noise
# are compared to analytical results from the known function.
#
# The frequency is calculated using the Octave 'filter' comma
# the coefficients are specified in REVERSE ORDER.
#
# Also note that for a filter of length 2M+1, 'filter' introd
# Compensation for this is needed to maintain correct alignme
#
clear
Npts = 125
noiselev = 0.08

# Generate original waveform, corrupted waveform, and analyti
puredat = zeros(1,Npts);
truedrv = zeros(1,Npts);
for iv=1:Npts
  arg= pi * iv * iv / 1100 + pi;
  puredat(iv) =  sin(arg) * iv*iv/10000;
  noisdat(iv) = puredat(iv) + (rand(1)-rand(1))*noiselev ;
  truedrv(iv) = pi * 2 * iv / 1100 * cos(arg) * iv*iv/10000 +
               sin(arg) * iv * 2 / 10000 ;
end

# Input signal
# figure(1)
# plot(1:Npts, puredat, 'b', 1:Npts, noisdat, 'r', 1:Npts, ze
# title("True and corrupted function data sequences")
# pause(3.0)

# Define the exact-fit and spectral-fit derivative estimators
# Note that the terms in the kernel for Octave's 'filter' comm
# For a filter of length 2M+1, the 'filter' function also shi
# M samples, which are compensated for plotting.
kernelx = [ -0.08333333   0.66666667  0.00000000  -0.66666667
kernells = [ -0.02874297    -0.01236535   0.07557966   0.1588
  -0.13561064    -0.15886784    -0.07557966   0.01236535  0.0

# Apply the two kinds of estimation filter to the noisy data
estdrvx = filter(kernelx, [1], noisdat);
estdrvlls = filter(kernells, [1], noisdat);
```

```
# Compare the results
figure(2)
plot(1:Npts-5, estdrvlls(6:Npts), 'g*',  1:Npts-2, estdrvx(3:N
set(gca, "linewidth", 2, "fontsize", 24)
xlabel("Function argument","fontsize",20,"fontweight","bold")
ylabel("Derivative value","fontsize",20,"fontweight","bold")
title("Comparing robust and exact polynomial derivative estima
legend("robust filter", "exact poly filter","analytical refere
```

--- minmax.m ---

```
% Solve for the filter coefficients of a min-max derivative e
% Reports the vector of coefficients and plot the response spe
% this operator.
%
% To construct the filter kernel, the calculated parameter va
% scaled by 1/2 and placed into the upper kernel locations; th
% the negatives of these are placed into the lower kernel loca
% The central term between them is always 0.0 from anti-symme

% Design paramaters:
%   Passband width (typically 0.05 to 0.20 of Nyquist limit)
%   Transition width (typically just a little wider than passb
%   Number of terms (final filter will be 2*nterms+1)
%   Error sensitivity in passband (typically 20 to 200, highe

% 13 term filter, half low band, peak 0.14735
%passb  = 0.16
%transition = 0.33
%nfreqs = 6
%psens  = 70.0
% 11 term filter, half low band, peak 0.14735
%passb  = 0.0680
%transition = 0.29
%nfreqs = 5
%psens  = 150.0
% 11 term filter, peak 0.08680
passb  = 0.175
transition = 0.31
nfreqs = 5
psens  = 28.0
% 9 term filter, peak 0.16143
```

```
%passb  = 0.18
%transition = 0.32
%nfreqs = 4
%psens  = 48.0

% Implementation parameter, resolution of waves in tableau
nsteps = 500
delfreq = pi/nsteps;

% Derived internal parameters
ipstart = 1;
ipend   = ceil(passb*nsteps);
nfit    = ipend-ipstart+1

isstart = ipend + floor(transition*nsteps) + 1
nstop   = nsteps-isstart
nterms  = nfit+nstop

% Matrix form: |-B   B   I   0  -m | * V  =  | -b  |
%              | B  -B   0   I  -m |         |  b  |
%
% Soln:    V = | +x -x  s1  s2 tol |'
%
% Cost:        | 0   0   0   0   1 |

% Rows in 'tableau' matrix
matrows = 2*nterms+1;
kcosts  = matrows;

% Columns in 'tableau' matrix
matcols = 2*nfreqs+2*nterms+2;
ktoler  = matcols-1;
kvalues = matcols;

% Construct the desired frequency response level for a
% derivative filter. Constant slope in fit band, 0 in
% top band, and transition does not matter so arbitrarily 0.
% Scale for sensitivity weighting.
bterms  = zeros(nterms,1);
for iterm=1:nfit
  bterms(iterm) = delfreq*iterm;
end

% Construct weighted basis terms by sampling sinewave functio
sinebasis = zeros(nterms,nfreqs);
for  kterm=1:nfreqs
```

```
  for  iterm=1:nfit
     sinebasis(iterm,kterm) = sin(kterm*iterm*delfreq);
  end
  for  iterm=1:nstop
     sinebasis(nfit+iterm,kterm) = sin(kterm*(isstart+iterm-1
  end
end
% Construct passband/stopband weights for tolerance column
mcol = [ones(nfit,1); ones(nstop,1)*psens];

% We now have enough information to construct the "tableau" ma
tableau = zeros(matrows,matcols);

% -- sine function basis columns, two copies of sine basis
tableau(1:nterms,1:nfreqs) = -sinebasis;
tableau(nterms+1:2*nterms,1:nfreqs) =  sinebasis;
tableau(1:nterms,nfreqs+1:2*nfreqs) =  sinebasis;
tableau(nterms+1:2*nterms,nfreqs+1:2*nfreqs) = -sinebasis;

% -- artificial slack columns
tableau(1:2*nterms,2*nfreqs+1:2*nfreqs+2*nterms) =  eye(2*nte

% -- tolerance bound column
tableau(1:nterms,ktoler) = -mcol;
tableau(nterms+1:2*nterms,ktoler) = -mcol;

% -- target values column
tableau(1:nterms,kvalues) =  -bterms;
tableau(nterms+1:2*nterms,kvalues) = bterms;

% -- relative cost row
tableau(kcosts,1:matcols)=zeros(1,matcols);
tableau(kcosts,ktoler)=1000;

# disp('Initial tableau:')
# tableau

% Bring the toler column into the basis. The M variable is re
% remain positive, so pivot on the element for which the ratio
% the value term to the M term is maximum negative. This estab
% an initial feasible basic solution.
highest = 0;
iprow = 0;
for  irow=1:matrows-1
  curtol = tableau(irow,ktoler);
  if  curtol ~= 0
```

```
    ratio = tableau(irow,kvalues)/tableau(irow,ktoler);
    if  ratio > highest
       lowest = ratio;
       iprow = irow;
    end
  end
end

# disp('Initial pivot to establish feasibility:')
iprow;
ipcol=ktoler;
tableau = mmpivot(tableau,iprow,ipcol);

while true
  % Find feasible pivot column
  ipcol = findpcol(tableau,nfreqs);
  if  ipcol < 1
     disp('No pivot column found!')
     break;
  end
  iprow = findprow(tableau,ipcol);
  disp('Pivot location:'), disp( [iprow,ipcol] );
  tableau = mmpivot(tableau,iprow,ipcol);
end

% Show the design results
params = mmresult(tableau,nfreqs);
figure(1)
mmplot
```

--- findpcol ---

```
function  pivcol = findpcol( oldmat, nfreq )
% Search the relative cost coefficients row for the largest n
% term and select its location as the pivot column. If none i
% return 0.  This is a subprogram used by the min-max design 
[rows,cols] = size(oldmat);

pivcol = 0;
smallest = 0;
for  icol=1:cols-1
  % Ingore tableau entries that are already close to zero
  if  abs(oldmat(rows,icol)) < 1.0e-8
     continue;
```

```
  end

  % Search supplemental relative cost coefficient row for sma
  if  oldmat(rows,icol) < smallest
    pivcol = icol;
    smallest = oldmat(rows,icol);
  end
end
```

--- findprow.m ---

```
function  pivrow = findprow( oldmat, pcol )
% Search the specified column of the matrix to determine the
% row to use for the next pivot operation using Dantiz's pivo
% rule.  Returns the index pivrow of the located row. This is
% a subprogam used by the min-max design process.

[rows,cols] = size(oldmat);

% Look for candidate pivot in all rows but last relative-cost
minval = 1.0e15;
pivrow = 1;
for  irow=1:rows-1
  if  oldmat(irow,pcol) > 0.0
    candidate = oldmat(irow,cols)/oldmat(irow,pcol);
    if  candidate < minval
       minval = candidate;
       pivrow = irow;
    end
  end
end
```

--- mmpivot.m ---

```
function  newmat = mmpivot( oldmat, iprow, ipcol )
% Perform a "pivot operation" at matrix location oldmat[prow,
% sets this value to 1.0 and clears all of the other terms in
% this column.  This is a subprogram used by the minmax estim

[rows,cols] = size(oldmat);
```

```
pivterm = oldmat(iprow,ipcol);
pivrow  = oldmat(iprow,1:cols);

newmat = zeros(rows,cols);
for  irow=1:rows
  if   irow==iprow
  % Special case for pivot row!
    newmat(irow,1:cols)=oldmat(irow,1:cols)*(1.0/pivterm);
    newmat(iprow,ipcol)=1.0;
  else
  % For other rows, determine the multiplier and perform pivo
    pivmult = -oldmat(irow,ipcol)/pivterm;
    newmat(irow,1:cols) = oldmat(irow,1:cols) + pivmult*pivro
    newmat(irow,ipcol)=0.0;
  end
end
```

--- mmplot.m ---

```
fit = zeros(nsteps);
halfsteps = floor(nsteps/2);
peak = 0;

# Observe the magnitude of peaks in the noise band
for  iterm=1:nsteps
   for kterm=1:nfreqs
     fit(iterm) = fit(iterm) + sin(kterm*iterm*delfreq)*param
   end
   afit = abs(fit(iterm));
   if  iterm>halfsteps && afit > peak
      peak = afit;
   end
end

# Generate reference response level line for plot
nideal = floor(nsteps*0.25);
ideal = zeros(1,nideal);
for iterm=1:nideal
  ideal(iterm) = pi*iterm/nsteps;
end

# Plot the min-max filter response characteristic
xscale = 1.0/nsteps;
```

```
"peak value" ; peak
plot( (1:nideal)*xscale, ideal,'g',  (1:nsteps-1)*xscale,abs(
set(gca, "linewidth", 2, "fontsize", 24)
xlabel("Frequency, fraction of Nyquist limit","fontsize",20,"
ylabel("Derivative value","fontsize",20,"fontweight","bold")
title("Min-max derivative estimator, 11-term filter")
```